torchex Documentation

Release 1.0.1

koji.ono

Feb 11, 2019

Contents:

CHAPTER 1

torchex API

1.1 Initialization

torchex.nn.init.**chrono_init**(*rnn: torch.nn.Module*, *Tmax=None*, *Tmin=1*)
chrono initialization(Ref: https://arxiv.org/abs/1804.11188)

torchex.nn.init.**feedforward_init**(*dnn: torch.nn.Module*, *init_mean=0*, *init_std=1*, *init_xavier: bool = True*, *init_normal: bool = True*, *init_gain: float = 1.0*)

torchex.nn.init.**rnn_init**(*rnn: torch.nn.Module*, *init_xavier: bool = True*, *init_normal: bool = True*, *init_gain: float = 1.0*, *init_mean: float = 0.0*, *init_std: float = 0.1*, *init_lower: float = 0.0*, *init_upper: float = 0.04*)

1.2 Utility

class torchex.nn.**PeriodicPad2d**(*pad_left: int = 0*, *pad_right: int = 0*, *pad_top: int = 0*, *pad_bottom: int = 0*)

Params torch.Tensor input Input(B, C, W, H)

https://github.com/ZichaoLong/aTEAM/blob/master/nn/functional/utils.py

class torchex.nn.**PeriodicPad3d**(*pad: int = 0*)
Only support isotropic padding

class torchex.nn.**Flatten**

1.3 Convlution

class torchex.nn.**MLPConv2d**(*self*, *in_channels*, *out_channels*, *ksize=None*, *stride=1*, *pad=0*, *activation=relu.relu*, *conv_init=None*, *bias_init=None*)
Two-dimensional MLP convolution layer of Network in Network. This is an "mlpconv" layer from the Network in Network paper. This layer is a two-dimensional convolution layer followed by 1x1 convolution layers and

interleaved activation functions. Note that it does not apply the activation function to the output of the last 1x1 convolution layer. Args:

in_channels (int or None): Number of channels of input arrays. If it is `None` or omitted, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.

out_channels (tuple of ints): Tuple of number of channels. The i-th integer indicates the number of filters of the i-th convolution.

ksize (int or pair of ints): Size of filters (a.k.a. kernels) of the first convolution layer. `ksize=k` and `ksize=(k, k)` are equivalent.

stride (int or pair of ints): Stride of filter applications at the first convolution layer. `stride=s` and `stride=(s, s)` are equivalent.

pad (int or pair of ints): Spatial padding width for input arrays at the first convolution layer. `pad=p` and `pad=(p, p)` are equivalent.

activation (callable): Activation function for internal hidden units. You can specify one of activation functions from built-in activation functions or your own function. It should not be an activation functions with parameters (i.e., `Link` instance). The function must accept one argument (the output from each child link), and return a value. Returned value must be a Variable derived from the input Variable to perform backpropagation on the variable. Note that this function is not applied to the output of this link. a keyword argument.

See: Network in Network. Attributes:

activation (callable): Activation function. See the description in the arguments for details.

class torchex.nn.**UpsampleConvLayer**(*in_channels*, *out_channels*, *kernel_size=1*, *stride=1*, *upsample=None*)

Upsamples the input and then does a convolution. This method gives better results compared to ConvTranspose2d. ref: http://distill.pub/2016/deconv-checkerboard/

1.4 Pooling

class torchex.nn.**GlobalAvgPool1d**

class torchex.nn.**GlobalAvgPool2d**

class torchex.nn.**GlobalMaxPool1d**

class torchex.nn.**GlobalMaxPool2d**

class torchex.nn.**MaxAvgPool2d**(*kernel_size*, *stride=None*, *padding=0*, *dilation=1*, *return_indices=False*, *ceil_mode=False*, *count_include_pad=True*)

Parameters

- **kernel_size** (*int or list*) – the size of the window to take a max and average over
- **stride** (*int or list*) – the size of stride to move kernel
- **padding** – implicit zero padding to be added on both sides
- **dilation** – a parameter that controls the stride of elements in the window
- **return_indices** – if True, will return the max indices along with the outputs. Useful when Unpooling later
- **ceil_mode** – when True, will use ceil instead of floor to compute the output shape

1.5 Cropping

class torchex.nn.**Crop2d**(*crop_left: int = 0*, *crop_right: int = 0*, *crop_top: int = 0*, *crop_bottom: int = 0*)

Params torch.Tensor input Input(B, C, W, H)

class torchex.nn.**Crop3d**(*crop_size*)

Params torch.Tensor input Input(B, C, D, W, H)

1.6 Local Convolution

class torchex.nn.**Conv2dLocal**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *in_size=None*, *padding=0*, *bias=True*)

1.7 Highway

class torchex.nn.**Highway**(*in_out_features*, *bias=True*, *activate=<class 'torch.nn.functional.relu'>*)

Highway module. In highway network, two gates are added to the ordinal non-linear transformation ($H(x) = activate(W_h x + b_h)$). One gate is the transform gate $T(x) = \sigma(W_t x + b_t)$, and the other is the carry gate $C(x)$. For simplicity, the author defined $C = 1 - T$. Highway module returns y defined as .. math:

```
y = activate(W_h x + b_h) \odot \sigma(W_t x + b_t) +
x \odot(1 - \sigma(W_t x + b_t))
```

The output array has the same spatial size as the input. In order to satisfy this, W_h and W_t must be square matrices. Args:

in_out_features (int): Dimension of input and output vectors. bias (bool): If `True`, then this function does use the bias. activate: Activation function of plain array. $tanh$ is also

available.

See: Highway Networks.

1.8 Inception

class torchex.nn.**Inception**(*in_channels*, *out1*, *proj3*, *out3*, *proj5*, *out5*, *proj_pool*)

Inception module of GoogLeNet. It applies four different functions to the input array and concatenates their outputs along the channel dimension. Three of them are 2D convolutions of sizes 1x1, 3x3 and 5x5. Convolution paths of 3x3 and 5x5 sizes have 1x1 convolutions (called projections) ahead of them. The other path consists of 1x1 convolution (projection) and 3x3 max pooling. The output array has the same spatial size as the input. In order to satisfy this, Inception module uses appropriate padding for each convolution and pooling. See: Going Deeper with Convolutions. Args:

in_channels (int or None): Number of channels of input arrays. out1 (int): Output size of 1x1 convolution path. proj3 (int): Projection size of 3x3 convolution path. out3 (int): Output size of 3x3 convolution path. proj5 (int): Projection size of 5x5 convolution path. out5 (int): Output size of 5x5 convolution path. proj_pool (int): Projection size of max pooling path.

TODO: Implement dimension reduction version.

class torchex.nn.**InceptionBN**(*in_channels*, *out1*, *proj3*, *out3*, *proj33*, *out33*, *pooltype='max'*, *proj_pool=None*, *stride=1*)

Inception module of the new GoogLeNet with BatchNormalization. This chain acts like *Inception*, while InceptionBN uses the BatchNormalization on top of each convolution, the 5x5 convolution path is replaced by two consecutive 3x3 convolution applications, and the pooling method is configurable. See: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. Args:

in_channels (int or None): Number of channels of input arrays. out1 (int): Output size of the 1x1 convolution path. proj3 (int): Projection size of the single 3x3 convolution path. out3 (int): Output size of the single 3x3 convolution path. proj33 (int): Projection size of the double 3x3 convolutions path. out33 (int): Output size of the double 3x3 convolutions path. pooltype (str): Pooling type. It must be either 'max' or 'avg'. proj_pool (int or None): Projection size in the pooling path. If

None, no projection is done.

stride (int): Stride parameter of the last convolution of each path.

See also:

Inception

1.9 Graph

class torchex.nn.**GraphLinear**(**argv*, ***kwargs*)

Graph Linear layer.

This function assumes its input is 3-dimensional. Differently from chainer.functions.linear, it applies an affine transformation to the third axis of input *x*.

See also:

torch.nn.Linear

class torchex.nn.**GraphConv**(*in_features*, *out_features*, *bias=True*)

Simple GCN layer, similar to https://arxiv.org/abs/1609.02907

class torchex.nn.**SparseMM**(*sparse*)

Sparse x dense matrix multiplication with autograd support. Implementation by Soumith Chintala: https://discuss.pytorch.org/t/ does-pytorch-support-autograd-on-sparse-matrix/6156/7

class torchex.nn.**GraphBatchNorm**(*num_features*)

Graph Batch Normalization layer.

See also:

torch.autograd.BatchNorm1d

class torchex.nn.**SequenceLinear**(*in_features*, *out_features*, *batch_first=True*, *nonlinearity='sigmoid'*, *init_mean=0*, *init_std=1*, *init_xavier: bool = True*, *init_normal: bool = True*, *init_gain=None*, *dropout=0.0*, *bias=True*)

x[T, B, F] -> nn.Linear(x[T, BxF]) -> x[T, B, F] if batch_first is True, x[B, T, F] -> x[T, B, F] -> nn.Linear(x[T, BxF]) -> x[T, B, F]

1.10 indrnn

class torchex.nn.**IndRNN**(*input_size*, *hidden_size*, *n_layer=1*, *batch_norm=False*, *step_size=None*, ***kwargs*)

Applies a multi-layer IndRNN with *tanh* or *ReLU* non-linearity to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$h_t = \tanh(w_{ih}x_t + b_{ih} + w_{hh}(*)h_{(t-1)})$$

where h_t is the hidden state at time *t*, and x_t is the hidden state of the previous layer at time *t* or $input_t$ for the first layer. (*) is element-wise multiplication. If `nonlinearity`='relu', then `ReLU` is used instead of *tanh*.

Args: input_size: The number of expected features in the input *x* hidden_size: The number of features in the hidden state *h* num_layers: Number of recurrent layers. nonlinearity: The non-linearity to use. Can be either 'tanh' or 'relu'. Default: 'tanh' bias: If `False`, then the layer does not use bias weights *b_ih* and *b_hh*.

Default: `True`

batch_first: If `True`, then the input and output tensors are provided as *(batch, seq, feature)*

Inputs: input, h_0

- **input** of shape *(seq_len, batch, input_size)*: tensor containing the features of the input sequence. The input can also be a packed variable length sequence. See `torch.nn.utils.rnn.pack_padded_sequence()` or `torch.nn.utils.rnn.pack_sequence()` for details.
- **h_0** of shape *(num_layers * num_directions, batch, hidden_size)*: tensor containing the initial hidden state for each element in the batch. Defaults to zero if not provided.

Outputs: output, h_n

- **output** of shape *(seq_len, batch, hidden_size * num_directions)*: tensor containing the output features (*h_k*) from the last layer of the RNN, for each *k*. If a `torch.nn.utils.rnn.PackedSequence` has been given as the input, the output will also be a packed sequence.
- **h_n** (num_layers * num_directions, batch, hidden_size): tensor containing the hidden state for *k* = *seq_len*.

Attributes: cells[k]: individual IndRNNCells containing the weights

Examples:

```
>>> rnn = nn.IndRNN(10, 20, 2)
>>> input = torch.randn(5, 3, 10)
>>> h0 = torch.randn(2, 3, 20)
>>> output = rnn(input, h0)
```

class torchex.nn.**IndRNNCell**(*input_size*, *hidden_size*, *bias=True*, *nonlinearity='relu'*, *hidden_min_abs=0*, *hidden_max_abs=None*)

An IndRNN cell with tanh or ReLU non-linearity. * https://arxiv.org/abs/1804.04849

$$h' = \tanh(w_{ih} * x + b_{ih} + w_{hh}(*)h)$$

With (*) being element-wise vector multiplication. If nonlinearity='relu', then ReLU is used in place of tanh.

Args: input_size: The number of expected features in the input x hidden_size: The number of features in the hidden state h bias: If `False`, then the layer does not use bias weights b_ih and b_hh.

Default: `True`

nonlinearity: The non-linearity to use ['tanh'|'relu']. Default: 'relu' hidden_min_abs: Minimal absolute inital value for hidden weights. Default: 0 hidden_max_abs: Maximal absolute inital value for hidden weights. Default: None

Inputs: input, hidden

- **input** (batch, input_size): tensor containing input features
- **hidden** (batch, hidden_size): tensor containing the initial hidden state for each element in the batch.

Outputs: h'

- **h'** (batch, hidden_size): tensor containing the next hidden state for each element in the batch

Attributes:

weight_ih: the learnable input-hidden weights, of shape *(input_size x hidden_size)*

weight_hh: the learnable hidden-hidden weights, of shape *(hidden_size)*

bias_ih: the learnable input-hidden bias, of shape *(hidden_size)*

Examples:

```
>>> rnn = nn.IndRNNCell(10, 20)
>>> input = Variable(torch.randn(6, 3, 10))
>>> hx = Variable(torch.randn(3, 20))
>>> output = []
>>> for i in range(6):
...     hx = rnn(input[i], hx)
...     output.append(hx)
```

class torchex.nn.**IndRNNTanhCell**

class torchex.nn.**IndRNNReLuCell**

1.11 Lazy Modules

These modules provide lazy evaluation for pytorch modules and help to create your neural network more easily and intuitively. You don't need pass input size to These modules. This Lazy evaluation comes from **chainer** which is one of the most powerful Deep-Learning framework.

class torchex.nn.**Linear**(*in_features*, *out_features=None*, *use_bias=True*, *xavier_init=True*)

Examples:

```
import torch
import torchex.nn as exnn

net = exnn.Linear(10)
# You don't need to give the size of input for this module.
# This network is equivalent to `nn.Linear(100, 10)`.

x = troch.randn(10, 100)
y = net(x)
```

class torchex.nn.**Conv1d**(*in_channels: int*, *out_channels: int*, *kernel_size: int = None*, *stride: int = 1*, *padding: int = 0*, *dilation: int = 1*, *groups: int = 1*, *bias: bool = True*, *xavier_init: bool = True*)

Parameters

- **out_channels** – the size of the window to take a max and average over
- **kernel_size** (*int or list*) – the size of the window to take a max and average over
- **stride** (*int or list*) – the size of stride to move kernel
- **padding** – implicit zero padding to be added on both sides
- **dilation** – a parameter that controls the stride of elements in the window
- **return_indices** – if True, will return the max indices along with the outputs. Useful when Unpooling later
- **ceil_mode** – when True, will use ceil instead of floor to compute the output shape

Examples:

```
import torch
import torchex.nn as exnn

net = exnn.Conv1d(10, 2)
# You don't need to give the size of input for this module.
# This network is equivalent to `nn.Conv1d(3, 10, 2)`.

x = troch.randn(10, 3, 28)
y = net(x)
```

class torchex.nn.**Conv2d**(*in_channels: int*, *out_channels: int*, *kernel_size: int = None*, *stride: int = 1*, *padding: int = 0*, *dilation: int = 1*, *groups: int = 1*, *bias: bool = True*, *xavier_init: bool = True*)

Examples:

```
import torch
import torchex.nn as exnn

net = exnn.Conv2d(10, 2)
# You don't need to give the size of input for this module.
# This network is equivalent to `nn.Conv2d(3, 10, 2)`.

x = troch.randn(10, 3, 28, 28)
y = net(x)
```

class torchex.nn.**Conv3d**(*in_channels: int*, *out_channels: int*, *kernel_size: int = None*, *stride: int = 1*, *padding: int = 0*, *dilation: int = 1*, *groups: int = 1*, *bias: bool = True*, *xavier_init: bool = True*)

Examples:

```
import torch
import torchex.nn as exnn

net = exnn.Conv3d(10, 2)
# You don't need to give the size of input for this module.
# This network is equivalent to `nn.Conv3d(3, 10, 2)`.

x = troch.randn(10, 3, 100, 28, 28)
y = net(x)
```

class torchex.nn.**LazyRNNBase**(*mode*, *hidden_size*, *num_layers=1*, *bias=True*, *batch_first=False*, *dropout=0*, *bidirectional=False*)

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index